Simulador de Supermercado

Ângelo Gomes – 201703990 – MIERSI
Eduardo Morgado – 201706894 – MIERSI
Laboratório de Programação(CC2013) 2018/2019
Trabalho de Grupo 2
FCUP

1. Introdução

1.1. Simulação de um Supermercado

É apresentada uma implementação de um programa de simulação do tempo de espera nas caixas de um supermercado aplicando um modelo de representação linear[1] para organização/estruturação de uma caixa.

Um supermercado é representado pelo **número de caixas** abertas durante o decorrer da simulação, a **afluência**[2] dos clientes ao supermercado, a **apetência**[3] de compras previstas para os clientes e o número de **ciclos**, ou seja, o tempo durante o qual o supermercado será simulado (novos clientes podem ser criados, dependendo do valor de afluência, são também verificados quais os clientes nas caixas cujas compras já foram todas processadas, ou seja, o cliente já finalizou o seu pedido e como tal, pode fazer *checkout* no supermercado).

Durante o decorrer da simulação, serão criados clientes e adicionados a uma das caixas disponíveis, para isso, deverá se considerada uma classe **cliente** que seja capaz de representar um cliente do supermercado[4].

Também deverá ser considerada uma classe **caixa** para representar uma caixa de supermercado, uma caixa de supermercado é representada por uma estrutura de dados linear (*queue* ou *priority queue*), por um número único que representa a caixa (**id**), o **instante** a partir do qual a caixa está pronta para atender um próximo cliente na fila, o número **total de clientes atendidos** pela caixa ao longo da simulação, o número **total de produtos processados** pela caixa ao longo da simulação, o **tempo total de espera** (sempre que um cliente é atendido, o tempo total de espera aumenta, baseado no tempo de espera do cliente: tempo atual – instante de *checkin* de cliente) e o número de **produtos processados por ciclo** pela caixa (este valor é um inteiro aleatório entre 1 e 5).

2. Estruturas Lineares

[1] Uma fila única de clientes ou duas filas de prioridade com prioridades normal e urgente.
[2] Um valor inteiro entre 1 (afluência mínima) e 100 (afluência máxima).
[3] Valor inteiro entre 1(apetência mínima) e 100 (apetência máxima). Quanto maior o valor de apetência, maior o número de compras feitas por cada cliente.
[4] Um cliente é representado pelo número de itens que deseja comprar e pelo instante/ciclo em que entrou no supermercado.

Nesta secção serão apresentadas as estruturas de dados lineares utilizadas para organizar os dados, sendo essas estruturas, uma **fila** normal e uma **fila de prioridades** (com duas prioridades, normal e urgente).

O programa de simulação de supermercado apresenta duas versões/ficheiros, uma primeira versão com uma implementação do supermercado recorrendo apenas a uma fila por cada caixa e outra versão tornando possível o uso de uma fila de 2 prioridades ou uma fila de uma prioridade apenas.

Para a implementação de ambas as filas, recorre-se a uma classe/módulo *Client.h* representando um cliente, a *Figura 1* apresenta o cabeçalho desse módulo, um cliente pode ser representado pelo número de itens que tenciona comprar e o instante de entrada no supermercado/caixa, a função *newClient* retorna um apontador para a posição em memória de um novo cliente.

```
#ifndef CLIENT_H
#define CLIENT_H

#define ITEMS(C) ((C)->items)
#define CHECKIN(C) ((C)->checkin_time)

//A client is defined by the number of items it buys and its time of arrival
typedef struct client{
    int items;
    int checkin_time;
}Client;

//Initializer
Client* newClient(int,int);

#endif
```

Figura 1- Client.h

2.1. Fila de Prioridade Única

Para representar uma fila de uma caixa de supermercado, basta considerar uma organização sequencial de elementos onde, o primeiro elemento a ser criado, a entrar na fila, será o primeiro elemento a ser retirado da fila, dessa forma, uma fila é uma estrutura de dados FIFO com duas operações básicas, *enqueue* e *dequeue*, operações de inserção e remoção de elementos, respetivamente.

A abordagem utilizada para implementar a extensão de uma fila pode suportada por vetores/*arrays* ou listas ligadas, ambas as formas apresentam as suas vantagens bem como as suas desvantagens.

2.1.1. Fila Suportada por Vetor

Numa implementação de uma fila suportada por um vetor, consideramos uma estrutura de tamanho fixo (com um tamanho máximo de expansão possível para a fila). Nesta abordagem, são guardados os índices do primeiro elemento da fila e do último elemento (existindo uma sequência contínua de elementos entre ambos), as inserções e remoções na fila realizam-se por deslocação dos índices de início/fim, uma vez que, esta estrutura é suportada por uma *array* de tamanho fixo, uma das vantagens é o custo de $O(1)$ para inserções/remoções, bem como a quantidade de espaço necessário para guardar o vetor, uma vez que, devido aos dados estarem representados em posições contíguas de memória, apenas é necessário armazenar um único apontador para a primeira posição do vetor, enquanto que, para uma fila suportada por listas ligadas, a quantidade de espaço necessário para representar a mesma informação é maior (é necessário espaço para mais que um apontador).

No entanto, esta abordagem encontra-se limitada pela sua falta de flexibilidade, numa fila de tamanho n, nunca será possível ter $n + 1$ elementos ao mesmo tempo na fila, para adicionar novos elementos caso esteja cheia, devem ser removidos outros elementos, a *Figura 2* apresenta uma representação de uma fila de inteiros suportada por um vetor com tamanho $n = 10$.

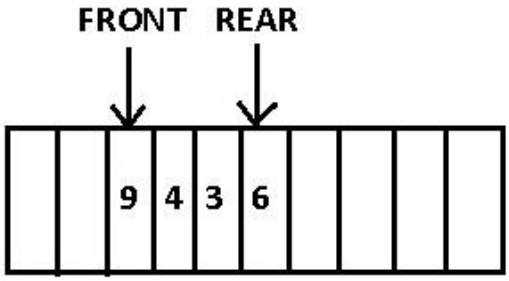

Figura 2- Fila suportada por um vetor

2.1.2. Fila Suportada por Listas Ligadas

Para uma implementação de uma fila suportada por uma lista ligada, a fila é vista como uma sequência de elementos ligados, a fila apresenta um apontador para o primeiro elemento e este por sua vez apresenta um apontador para o elemento seguinte, este elemento pode existir ou não.

Para remover um elemento, *dequeue*, basta deslocar o apontador para o próximo elemento, isto é, o apontador para o primeiro elemento passa agora a apontar para o elemento a seguir ao primeiro, esta operação tem custo linear. No entanto, para adicionar um elemento, uma vez que este deve ser adicionado ao fim da fila requer uma deslocação na lista podendo ter uma complexidade $O(n)$ para uma fila com n elementos. Para evitar este problema, uma fila com suporte de listas ligadas pode também guardar um apontador para o último elemento válido, não nulo, na fila (no momento de criação da lista e até que esta tenha pelo menos 2 elementos, ambos os apontadores apontam para a mesma posição em memória) dessa forma, a operação de adicionar um elemento à fila é uma operação de complexidade linear, nessa operação, o último elemento passa a apontar para o novo elemento e o apontador para o fim da lista

passa agora a apontar para o novo elemento adicionado à lista. A *Figura 3* apresenta um esquema de uma fila suportada por listas ligadas.

Apesar desta abordagem resolver o problema de flexibilidade da abordagem anterior, apresenta um custo maior de memória, para a implementação anterior apenas era necessário um apontador para a primeira posição em memória do vetor, no entanto, para esta noa abordagem são necessários $n + 2$ apontadores para uma fila de tamanho n, são necessários 2 apontadores para o primeiro e o último elemento bem como n apontadores para o próximo elemento.

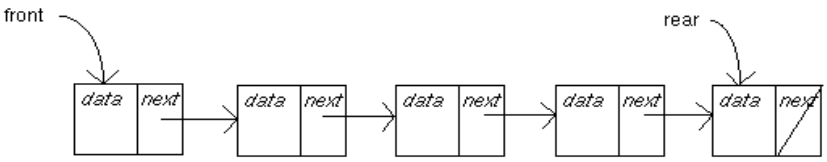

Figura 3- Fila suportada por lista ligada

Esta abordagem, devido à sua flexibilidade, é a implementação de suporte para uma fila, representando uma caixa (uma fila de elementos, onde cada elemento é do tipo *Client*). As *Figuras 4* e 5 apresentam o cabeçalho deste módulo.

```c
#include "Client.h"
#ifndef QUEUE_H
#define QUEUE_H

/*--------------------------------------------------------------------------------
|                          Macro Definition                                      |
--------------------------------------------------------------------------------*/
#define CLIENT(QE) ((QE)->client)
#define NEXT(QE) ((QE)->next)

#define FIRST(Q) ((Q)->first)
#define LAST(Q) ((Q)->last)
#define SIZE(Q) ((Q)->size)


/*--------------------------------------------------------------------------------
|                          Queue Element                                         |
--------------------------------------------------------------------------------*/
typedef struct queuelement{
    Client* client;
    struct queuelement* next;
}QueueElem;


/*--------------------------------------------------------------------------------
|                          Queue Definition                                      |
--------------------------------------------------------------------------------*/
typedef struct queue{
    QueueElem* first;
    QueueElem* last;
    int size;
}Queue;
```

Figura 4- Queue.h

```
/*-------------------------------------------------------------------------
|                          Queue Functions                                |
-------------------------------------------------------------------------*/
Queue* createQueue();

int isQueueEmpty(Queue*);
int size(Queue*);
Client* getFirst(Queue*);
Client* getLast(Queue*);

void enqueue(Client*,Queue*);
Client* dequeue(Queue*);

void printQueue(Queue*);

/*-------------------------------Free--------------------------------------*/
void freeQueue(Queue*);
void freeQueueElem(QueueElem*,QueueElem*);

#endif
```

Figura 5- Queue.h

Este módulo considera uma *Queue* como sendo dois apontadores, um para o primeiro elemento da fila e outro para o último elemento da fila, bem como o tamanho da fila. Os elementos da fila *QueueElement* apresentam um apontador para o cliente que representam bem como um apontador para o próximo elemento na lista.

2.2. Fila de Prioridades

Partindo da ideia de uma fila (neste caso suportada por listas ligadas) de prioridade única, ou seja, apenas existe uma fila para guardar os dados, considera-se agora a possibilidade de existirem mais filas, podendo estas ser organizadas por prioridades, ou seja, elementos são adicionados à fila corresponde à prioridade do elemento.

A fila de prioridades implementada tem duas prioridades, normal e urgente, os elementos/clientes são adicionados à fila indicando a sua prioridade e são colocados no fim da fila respetiva, elementos são removidos da fila através da remoção do primeiro elemento na fila urgente, caso esta não tenha elementos, então será removido o primeiro elemento da fila normal. Em filas de prioridade é possível aumentar a prioridade de um elemento, ou seja, remover o primeiro elemento da fila normal e adicioná-lo ao final da fila urgente.

A *Figura 6* apresenta o cabeçalho deste módulo, sendo este suportado por sua vez pelo módulo/cabeçalho *Queue.h*.

```c
#include "Client.h"
#include "Queue.h"
#ifndef PRIORITY_QUEUE_H
#define PRIORITY_QUEUE_H

/*--------------------------------------------------------------------------
|                          Macro Definition                                |
--------------------------------------------------------------------------*/
#define URGENT(PQ) ((PQ)->urgent)
#define NORMAL(PQ) ((PQ)->normal)


/*--------------------------------------------------------------------------
|                          PQueue Definition                               |
--------------------------------------------------------------------------*/
typedef struct priorityqueue{
    Queue* urgent;
    Queue* normal;
}PriorityQueue;


/*--------------------------------------------------------------------------
|                          PQueue Functions                                |
--------------------------------------------------------------------------*/
PriorityQueue* newPriorityQueue();

void addElem(Client*,int,PriorityQueue*); //enqueue in queue of priority int
Client* dequeuePQ(PriorityQueue*); //Removes first element with highest priority
void raiseUrgency(PriorityQueue*);//Increases priority of first element in Normal
Client* serviceElement(PriorityQueue*); //Gets element with highest priority

int isPQueueEmpty(PriorityQueue*);

void printPriorityQueue(PriorityQueue*);

void freePQueue(PriorityQueue*);

#endif
```

Figura 6- PriorityQueue.h

3. Uma Caixa de Supermercado

Uma caixa de supermercado deverá possuir um número de identificação (um inteiro), deverá armazenar o número total de clientes que foram atendidos[5], o número total de produtos processados, o tempo geral de espera, o tempo/instante para o qual a caixa pode atender um próximo cliente, deverá também definir uma capacidade de processamento de produtos por unidade de tempo/por ciclo.

[5] Um cliente é atendido quando todos os seus itens tiverem sido processados pela caixa.

Serão apresentadas duas figuras, ambas representando cabeçalhos de uma caixa de supermercado, um dos cabeçalhos é de uma primeira versão (ficheiro *Supermarket_With_Queue*) onde uma caixa de supermercado é representada apenas por uma única fila, enquanto que, o outro cabeçalho representa uma outra versão de uma caixa (ficheiro *Supermarket_With_Queue_and_PQueue*) onde uma caixa de supermercado pode ser representada por uma fila única ou por uma fila de prioridades, dependo da escolha do utilizador do programa.

```c
#include "Queue.h"
#include "Client.h"

#ifndef CASHIER_H
#define CASHIER_H

/*-----------------------------------------------------------------------------
|                          Macro Definition                                   |
-----------------------------------------------------------------------------*/
#define ID(C) ((C)->cashier_id)             //Id of cashier C
#define LINE(C) ((C)->cashier_line)
#define CLIENTS(C) ((C)->total_clients)     //#clients serviced in cashier C
#define PRODUCTS(C) ((C)->total_products)   //#products scanned in cashier C
#define WAIT_TIME(C) ((C)->total_waiting_time) //total wait time for cashier C
#define NEXT_SERVICE(C) ((C)->servicing_available_time)
#define SCAN_POWER(C) ((C)->scanning_power)//Number of products the cashier scans


/*-----------------------------------------------------------------------------
|                          Cashier Definition                                 |
-----------------------------------------------------------------------------*/
typedef struct cashier{
    Queue* cashier_line;
    int cashier_id;
    int total_clients;
    int total_products;
    int total_waiting_time;
    int servicing_available_time;
    int scanning_power;
}Cashier;


/*-----------------------------------------------------------------------------
|                          Cashier Functions                                  |
-----------------------------------------------------------------------------*/
Cashier* openCashier(int,int);

void delayServiceTo(int,Cashier*);    //Updates servicing_available_time
void processedProduct(int,Cashier*);  //Updates total_products
void timeIncrease(int,Cashier*);      //Updates total_waiting_time
int isCashierEmpty(Cashier*);

void clientCheckin(Client*,Cashier*);
void clientCheckout(Cashier*); //removes client and updates total_clients

void printCashier(Cashier*);

void serviceClient(int,Cashier*);
/*-----------------------------Free--------------------------------------------*/
void closeCashier(Cashier*);

#endif
```

Figura 7- Cashier.h Supermarket_With_Queue

```c
#include "Client.h"
#include "Queue.h"
#include "PriorityQueue.h"

#ifndef CASHIER_H
#define CASHIER_H

/*-----------------------------------------------------------------------------
|                          Macro Definition                                   |
-----------------------------------------------------------------------------*/
#define TYPE(C) ((C)->type)                 //Type of cashier C
#define PQ(C) ((C)->line.cashier_pqueue)    //Get priority queue of cashier C
#define Q(C) ((C)->line.cashier_queue)      //Get queue of cashier C

#define ID(C) ((C)->cashier_id)             //Id of cashier C
#define CLIENTS(C) ((C)->total_clients)     //#clients serviced in cashier C
#define PRODUCTS(C) ((C)->total_products)   //#products scanned in cashier C
#define WAIT_TIME(C) ((C)->total_waiting_time) //total wait time for cashier C
#define NEXT_SERVICE(C) ((C)->servicing_available_time)
#define SCAN_POWER(C) ((C)->scanning_power)//Number of products the cashier scans


/*-----------------------------------------------------------------------------
|                        Cashier Type Definition                              |
-----------------------------------------------------------------------------*/
typedef enum que_t{
    QUEUE,PQUEUE
}Type;


/*-----------------------------------------------------------------------------
|                          Cashier Definition                                 |
-----------------------------------------------------------------------------*/
typedef struct cashier{
    Type type;
    union{
        Queue* cashier_queue;
        PriorityQueue* cashier_pqueue;
    }line;
    int cashier_id;
    int total_clients;
    int total_products;
    int total_waiting_time;
    int servicing_available_time;
    int scanning_power;
}Cashier;
```

Figura 8- Cashier.h Supermarket_With_Queue_and_PQueue

```
/*-------------------------------------------------------------------------------
|                          Cashier Functions                                    |
-------------------------------------------------------------------------------*/
Cashier* openCashier(int,int,int);

void delayServiceTo(int,Cashier*);    //Updates servicing_available_time
void processedProduct(int,Cashier*);  //Updates total_products
void timeIncrease(int,Cashier*);      //Updates total_waiting_time
int isCashierEmpty(Cashier*);

void clientCheckin(Client*,int,Cashier*);
void clientCheckout(Cashier*);

void printCashier(Cashier*);

void serviceClient(int,Cashier*);     //Service the first client in the cashier

/*------------------------------Free-----------------------------------------*/
void closeCashier(Cashier*);

#endif
```

Figura 9- Cashier.h Supermarket_With_Queue_and_PQueue

4. Um Supermercado

Um supermercado pode ser representado por um agrupamento de caixas, capaz de atender clientes, isto é, percorrer cada caixa onde, cada caixa irá possivelmente atender um cliente, sendo assim, um supermercado pode ser uma estrutura de um apontador para um "vetor" de caixas, isto é, uma sequência de caixas. As *Figuras 10* e *11* apresentam o cabeçalho para este módulo, este cabeçalho pode ser encontrado no ficheiro *SupermarketSimulator/Supermarket_With_Queue_and_PQueue*.

```
#include "Cashier.h"
#ifndef SUPERMARKET_H
#define SUPERMARKET_H

/*-------------------------------------------------------------------------------
|                          Macro Definition                                     |
-------------------------------------------------------------------------------*/
#define CASHIERS(SM) ((SM)->cashiers_list)
#define NCASHIERS(SM) ((SM)->n_cashiers)


/*-------------------------------------------------------------------------------
|                          Supermarket Definition                               |
-------------------------------------------------------------------------------*/
//A supermarket will contain the number of open cashiers and its list/array
typedef struct supermarket{
    Cashier** cashiers_list;
    int n_cashiers;
}Supermarket;
```

Figura 10- Supermarket.h

```
/*-------------------------------------------------------------------------
|                          Supermarket Functions                          |
-------------------------------------------------------------------------*/
Supermarket* openSupermarket(int,int,int);

int isSupermarketInUse(Supermarket*);
void printSupermarket(Supermarket*);

void serviceClients(int,Supermarket*); //Service clients in the cashiers
void closingOfAccounts(Supermarket*); //Calculates the usage of all the cashiers

void closeSupermarket(Supermarket*);

#endif
```

Figura 11- Supermarket.h

Nota Adicional: Este relatório, bem como o programa utilizado para a simulação de um supermercado podem ser encontrados em https://github.com/eamorgado/A-Basic-Supermarket-Simulator.git. O repositório contém dois ficheiros, *SupermarketSimulator* (representando o simulador de um supermercado com as duas opções, de uma implementação apenas com filas e outra com implementação usando filas ou filas de prioridade) e *PriorityQueue* contendo as versões base de uma fila e uma fila de prioridades para inteiros.